INDIUM

Building Platform-Agnostic Scalable Hybrid Search

Semantic and keyword retrieval strategies for modern enterprise search.

Whitepaper

Written by
Ashish Kumar
Chief AI Architect

Introduction

Search has become a core part of digital products, whether it is helping patients find the right doctor or helping shoppers narrow down millions of products quickly. The difference between effective and ineffective search often determines whether users find what they need quickly or leave the product altogether.

Traditional keyword search worked well when datasets were smaller and queries were predictable. Exact-match retrieval struggles when user intent and document language do not align. Semantic search improves intent understanding, but it can miss highly specific identifiers, filters, and exact-match requirements.

Hybrid search combines both approaches by running keyword and semantic retrieval together. Most production search systems now rely on this model to balance precision and relevance. This paper examines the architecture and operational patterns behind production-ready hybrid search systems.

1. Why Hybrid Search – Production Search Scenarios

The Fundamental Tension

Keyword search (BM25) retrieves results by matching exact or closely related terms. It works well for highly specific queries but struggles when users describe something differently from how it appears in the index. A search for “heart doctor” may not return results for “cardiologist” unless synonym mapping is explicitly configured.

Semantic search approaches the problem differently. Queries and documents are converted into vector embeddings, allowing the system to retrieve results based on meaning rather than exact wording. This improves intent understanding, especially when user language varies. However, semantic retrieval can miss precision-heavy queries such as model numbers, drug names, or diagnostic codes where exact matches matter more than contextual similarity.

Hybrid search combines both retrieval methods and merges the results into a single ranked list. This balance between semantic understanding and exact-match retrieval makes hybrid search more effective for production workloads.

Healthcare: Finding the Right Provider

Consider a provider search system for a health plan. A member might search:

"endocrinologist near me accepting new patients who speaks Spanish"

This query has four distinct components: a specialty (semantic – "endocrinologist" maps to "hormone specialist," "diabetes doctor," etc.), a location constraint (geo-filter), a status filter (accepting patients), and a language preference (exact match).

A keyword-only search would fail on the semantic specialty lookup. Vector search, on the other hand, can struggle with strict filters and may surface a Portuguese-speaking endocrinologist 50 miles away because the embedding is semantically similar.

Other healthcare queries where hybrid search becomes critical:

Query	Why hybrid matters
"doctor who treats GERD"	Keyword: "GERD"; Semantic: maps to "gastroenterologist," "acid reflux specialist"
"pediatric cardiologist Stanford"	Exact institution match (keyword) + specialty understanding (semantic)
"find an in-network therapist for anxiety"	Network filter (keyword/filter) + specialty understanding (semantic)
"NPI 1234567890"	Pure exact match — keyword wins, semantic adds no value

E-commerce: Navigating the Catalog

An online retailer with 10 million SKUs faces a different class of problem. A search for:

"lightweight laptop for video editing under $1500"

requires semantic understanding ("lightweight" maps to weight specifications, "video editing" maps to GPU class and RAM), numerical range filtering ($1500), and category awareness.

Keyword search returns every product description containing “lightweight,” which could include unrelated items like camera bags. Semantic search may surface a $2000 workstation simply because it is contextually related to “video editing.”

A few e-commerce examples:

Query	Failure mode without hybrid
"Nike Air Max 270 size 10"	Semantic misses the exact model; keyword misses related colorways
"gift for a runner who has everything"	Pure keyword returns nothing; semantic can surface curated gift sets
"same as last order"	Requires personalization context + exact product recall
"eco-friendly yoga mat non-slip"	"Non-slip" is a critical keyword; "eco-friendly" is semantic

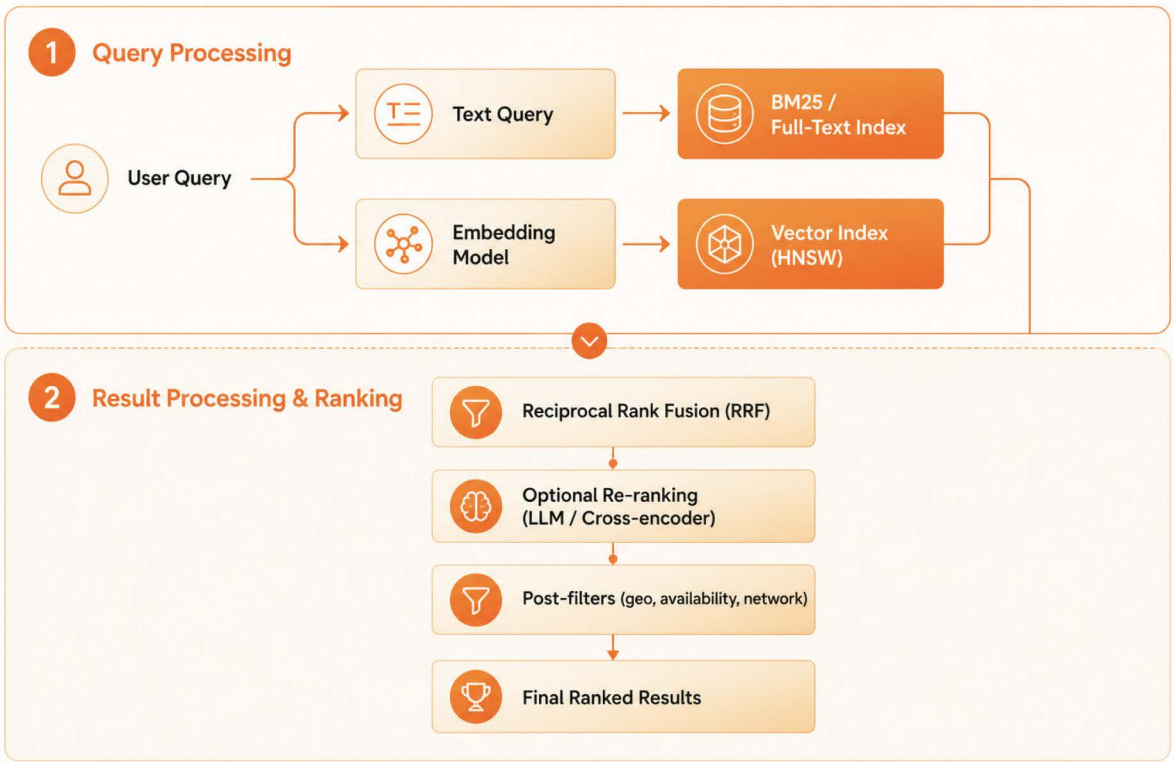
The General Principle

The queries where hybrid search matters most share a common structure: they combine **precise constraints** (model numbers, dates, codes, prices, locations) with **conceptual intent** (specialty, use case, mood, purpose). Any system that handles only one side of this spectrum will fail a meaningful fraction of real-world queries.

2. High-Level Architecture: Semantic + Keyword + RRF

The Retrieval Pipeline

A production hybrid search system runs keyword and vector retrieval in parallel against the same corpus, then merges the results using a scoring function before applying any post-retrieval steps.



BM25 – Keyword Retrieval

BM25 (Best Match 25) is a probabilistic ranking function that scores documents based on term frequency, inverse document frequency, and document length normalization. It is the default algorithm in Elasticsearch, Solr, OpenSearch, and Azure AI Search's full-text mode.

For a query term q and document d:

$$\text{BM25}(q, d) = \sum \text{IDF}(q_i) \cdot \left(f(q_i, d) \cdot (k1 + 1) \right) / \left(f(q_i, d) + k1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)\right)$$

Where $f(q_i, d)$ is term frequency, $|d|$ is document length, and k1 and b are tuning parameters (typically 1.2 and 0.75).

Vector Search – Semantic Retrieval

Documents are embedded into a dense vector space using an embedding model (OpenAI text-embedding-3, BioBERT, E5, BGE, etc.). At query time, the query is embedded using the same model and approximate nearest neighbor (ANN) search finds the k most similar vectors. HNSW (Hierarchical Navigable Small World) is the dominant ANN graph algorithm – it achieves O(log N) search time with recall rates above 95% at typical configurations.

Reciprocal Rank Fusion (RRF)

RRF is a rank aggregation method that combines ranked lists without requiring normalized scores across systems. For a document d appearing in ranked list r:

$$RRF\_score(d) = \sum_r 1 / (k + rank_r(d))$$

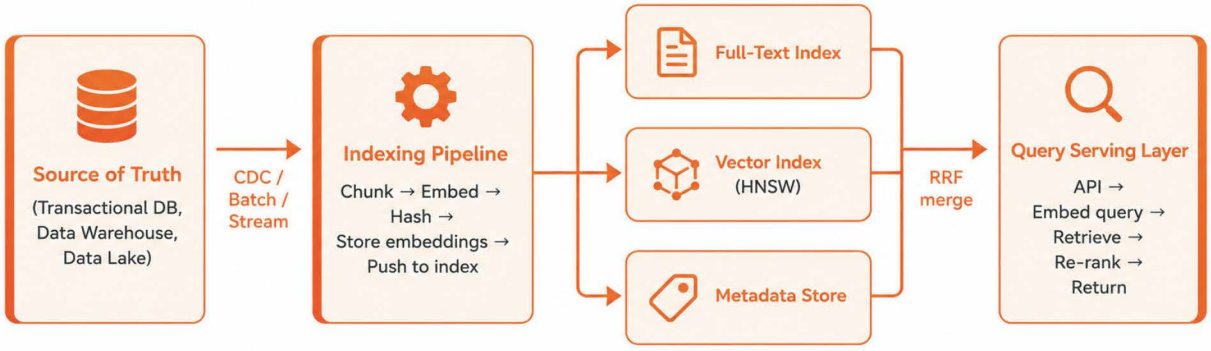
Where k is a constant (typically 60) that controls the influence of lower-ranked documents. RRF is robust, does not require score calibration between BM25 and vector systems, and consistently outperforms simple score interpolation in benchmarks.

Optional: Re-ranking

For latency-tolerant use cases, a cross-encoder re-ranker (Cohere Rerank, BGE-Reranker, or a fine-tuned BERT model) can re-score the top-K merged results using full query-document attention. This significantly improves precision at the cost of latency – typically used for the top 20–50 candidates.

3. Reference Architecture Across Cloud Platforms

A well-designed hybrid search system separates four concerns: the source of truth, the indexing pipeline, the vector/search index, and the query serving layer. Each major cloud platform provides native primitives for all four, with different tradeoffs.



Azure, AWS and GCP – Combined Reference

The table below outlines the key architectural layers for hybrid search on Azure, AWS, and GCP, along with the role each layer plays in the overall system.

Databricks and cloud-agnostic architectures are covered separately.

Layer	What it does	Azure	AWS	GCP
Source of Truth	The authoritative database that owns the raw provider or product data. The search index is always downstream of this and never the source of truth itself.	Azure SQL / Cosmos DB	Amazon RDS / DynamoDB / S3	Cloud SQL / Firestore / BigQuery
Embedding Generation	Converts text fields (specialty, description, name) into dense vector representations using a language model. Runs at index time, and optionally at query time.	Azure OpenAI (text-embedding-3)	Amazon Bedrock (Titan Embeddings, Cohere)	Vertex AI (text-embedding-gecko, text-multiingual-embedding)
Embedding Persistence	Stores generated embeddings durably so they can be reused on index rebuilds without re-calling the (expensive) embedding model API. Keyed by document ID + content hash.	Azure Blob Storage	Amazon S3	Cloud Storage
Full-Text Index (BM25)	Inverted index for keyword search. Scores documents by term frequency and inverse document frequency. Fast, deterministic, exact-match friendly.	Azure AI Search	Amazon OpenSearch Service	Vertex AI Search (formerly Enterprise Search)
Vector Index (HNSW)	Approximate nearest neighbor (ANN) graph for semantic search. Finds vectors geometrically close to the query embedding. HNSW achieves O(log N) search time at >95% recall.	Azure AI Search (integrated — same service as BM25)	Amazon OpenSearch k-NN or Bedrock Knowledge Bases	Vertex AI Vector Search (ScaNN-based)
Indexing Pipeline	Orchestrates the flow from source data → chunking → embedding → hash check → push to search index. Can be scheduled (batch) or event-driven (streaming).	AI Search Indexer (pull) or Azure Container Apps (push)	AWS Lambda / AWS Glue / Amazon EventBridge	Cloud Dataflow / Cloud Functions
Change Detection / CDC	Identifies which records have changed since the last indexing run. Critical for incremental indexing → avoids full corpus rebuilds on every update.	SQL Change Tracking / Cosmos DB Change Feed	DynamoDB Streams / RDS Change Data Capture	Pub/Sub + Datastream (CDC from Cloud SQL)
Query Serving	The application layer that receives search requests, embeds the query, calls both BM25 and vector indexes in parallel, applies RRF, and returns ranked results.	Azure Container Apps + Azure API Management	AWS Lambda + Amazon API Gateway	Cloud Run + Apigee
Cache	Stores recent query embeddings and/or full result sets to avoid redundant embedding API calls and repeated index lookups for popular queries. TTL-based expiry.	Azure Cache for Redis	Amazon ElastiCache (Redis)	Memorystore (Redis)
Auth / Security	Manages API authentication, secrets rotation, and RBAC. Ensures tenant isolation at the request level and secure service-to-service communication via managed identities.	Azure Entra ID + Key Vault	AWS IAM + Secrets Manager	Google IAM + Secret Manager
Observability	Monitoring, logging, tracing, and alerting across the entire pipeline for reliability, performance, and cost optimization.	Azure Monitor + Application Insights	Amazon CloudWatch + AWS X-Ray	Cloud Monitoring + Cloud Trace
Cost Optimization	Controls costs across embedding, storage, indexing, and serving through right-sizing, caching, TTLs, lifecycle policies, and usage alerts.	Azure Cost Management + Advisor	AWS Cost Explorer + Trusted Advisor	Cloud Billing + Recommender

Azure AI Search is unique in providing BM25 and HNSW in a single managed service with native RRF – one index, one SDK, one billing line for both retrieval modes.

OpenSearch on AWS supports hybrid search natively via its hybrid query type and neural search plugin. Bedrock Knowledge Bases provides a higher-level RAG abstraction handling chunking, embedding, and vector storage automatically.

GCP's Vertex AI Vector Search (previously Matching Engine) uses Google's ScaNN algorithm and offers extremely high throughput at scale. For teams wanting everything in one relational system at < 10M vectors, pgvector on Cloud SQL is the simplest alternative.

Alternative – dedicated vector DB on AWS: Replace OpenSearch k-NN with Pinecone, Weaviate, or Qdrant on EC2/EKS. Use OpenSearch purely for BM25 full-text. Merge results via RRF in the application layer.

Alternative – pgvector on Cloud SQL (GCP): For moderate scale (< 10M vectors), PostgreSQL with the pgvector extension eliminates a separate vector database. Full-text search uses PostgreSQL's built-in tsvector.

Databricks

Databricks is increasingly relevant for teams whose data already lives in the Lakehouse and who want to avoid moving data to a separate search system.

Component	Databricks / Delta Lake Service
Source of truth	Delta Lake tables
Embedding generation	Databricks Model Serving (BGE, E5, custom fine-tuned)
Embedding persistence	Delta Lake (vector column in bronze/silver table)
Full-text index	Delta Lake full-text search (preview) or external Elasticsearch
Vector index	Databricks Vector Search (managed HNSW on Delta tables)
Indexing pipeline	Delta Live Tables (DLT) streaming pipeline
Change detection	Delta Lake Change Data Feed (CDF)
Query serving	Databricks Model Serving endpoint or external API
Cache	External Redis

> Databricks Vector Search, released in 2024, allows you to create a vector index directly on a Delta table column. When the underlying Delta table is updated, the index is automatically synchronized using Delta's Change Data Feed. This is one of the most streamlined approaches to incremental indexing, with no separate pipeline or custom change detection logic required.

Cloud-Agnostic: Dedicated Vector Database Flavor

For teams that need portability or are operating across multiple clouds:

Component	Tool
Full-text search	Elasticsearch (self-hosted or Elastic Cloud)
Vector database	Qdrant (self hosted on k8s) or Pinecone
Embedding	Any model via HuggingFace / OpenAI API
Indexing pipeline	Apache Airflow or Prefect
Change detection	Debezium (CDC from any relational DB)
Query serving	FastAPI on Kubernetes
Cache	Redis

This stack runs identically on any cloud and avoids proprietary lock-in. The tradeoff is higher operational burden, you own the Qdrant cluster, the Elasticsearch cluster, and the Airflow deployment.

Managed Search vs Dedicated Vector Database – When to Choose What

One of the most consequential architectural decisions in a hybrid search system is whether to use the managed search service bundled with your cloud platform (Azure AI Search, OpenSearch, Vertex AI Search) or to deploy a dedicated, purpose-built vector database (Qdrant, Pinecone, Weaviate). There is no universal right answer and the decision depends on scale, team capacity, and how much of the stack you want to own.

Dimension	Managed Search Service (Azure AI Search / OpenSearch / Vertex AI)	Dedicated Vector DB (Qdrant / Pinecone / Weaviate)
Setup & ops burden	Low: Fully managed, no cluster to run. Deploy via portal or Terraform in minutes.	Higher: You provision, monitor, upgrade, and scale the cluster yourself (or pay Pinecone/Qdrant Cloud for managed hosting).
Hybrid search support	Native: BM25 + vector in one service, RRF built in. No stitching required.	Vector search only (primary): Full-text search requires a separate Elasticsearch instance. RRF must be implemented in your application layer.
HNSW parameter control	Limited: Managed services expose fewer knobs (m, efConstruction, efSearch) than self-hosted alternatives.	Full control: HNSW graph parameters, quantization settings, and segment configuration. Qdrant exposes the most tuning options of any vector DB.
Vector quantization	Supported: By Azure AI Search (scalar + binary) Reduces memory 4x–32x. Less granular than self-hosted.	Fine-grained: Qdrant supports scalar, product, and binary quantization with per-collection oversampling settings.
Multi-tenancy	Via index-per-tenant or filter-based namespacing. Index sprawl is a real concern at 100+ tenants.	Qdrant collections & named vectors: Natively support multi-tenancy. Payload filtering is extremely efficient, making shared-index multi-tenancy practical at scale.
Cost at small-mid scale (< 50M vectors)	Competitive: Managed premium justified by zero ops overhead. Azure AI Search S1 ≈ $250/month.	Qdrant Cloud from ~$200/month for production: Self-hosted on AKS/EKS ~$400–600/month (compute + storage). Cost parity or slightly higher.
Cost at large scale (> 100M vectors)	Steeper: Pricing scales sharply with partitions and replicas. Azure AI Search S3 × 4 partitions ≈ $5,800/month.	Significantly cheaper: You pay for raw compute, not a managed search premium. Self-hosted Qdrant on AKS at 100M vectors ≈ $800–1,200/month.
Latency at high QPS	Good: Managed infrastructure handles scaling. Latency increases under heavy concurrent load without adding replicas (cost increases).	Excellent: Qdrant's Rust-based engine is optimized for sub-millisecond ANN queries. Better raw throughput than managed services at equivalent cost.
Cloud portability	None: Tightly coupled to the cloud provider. Migrating means full re-index.	Full: Qdrant, Weaviate, and Pinecone run on any cloud or on-prem. Docker image is identical across environments.
Ecosystem integration	Deep: Native connectors to Azure OpenAI, Bedrock, Vertex AI. First-class LangChain and Semantic Kernel support.	Good: LangChain, LlamaIndex, and DSPy all support Qdrant, Pinecone, and Weaviate. Integration requires slightly more configuration.
Best for	Teams that want operational simplicity, are already on one cloud, and have < 100M vectors. Especially strong when BM25 + vector in a single service matters.	Teams at large vector scale (100M+), multi-cloud or cloud-agnostic requirements, or who need maximum control over HNSW tuning and quantization.

Recommendation: Start with your cloud's managed service for speed to value. At 50–100M+ vectors, model the cost delta – if self-hosted Qdrant saves > $3,000/month, the operational overhead of a managed Kubernetes cluster becomes worth it. Never migrate mid-project; make the decision upfront.

Combined Cross-Platform Reference Table

The table on the following page consolidates all five platforms side by side for quick reference.

(Also useful as a starting point for Excalidraw architecture diagrams.)

4. Incremental Indexing and CDC Strategies

The Core Problem

Re-indexing an entire corpus from scratch after every update becomes expensive at scale. A provider directory with 1 million records, each requiring an embedding API call, would cost thousands of dollars and take hours to rebuild fully. Production systems need to identify and process only changed records.

Strategy 1 – High Water Mark (Polling)

The simplest approach. A last modified or updated_at timestamp column on the source table is used as a watermark. The indexer polls periodically, fetching only rows where [updated_at > last_run_timestamp].

- **Limitations**: Soft deletes must be handled explicitly (tombstone records). Clock skew between writer and reader can cause missed records. Hard deletes are invisible.

- **Best for**: Azure SQL with Azure AI Search Indexer (native support via "HighWaterMarkChangeDetectionPolicy"), simple relational sources.

Strategy 2 – SQL Change Tracking / CDC

Database-native change capture provides a complete, ordered log of all inserts, updates, and deletes, including hard deletes.

- **SQL Server / Azure SQL**: "CHANGE_TRACKING" or "CDC" feature built in. Azure AI Search "SqlIntegratedChangeTrackingPolicy" consumes this natively.

- **PostgreSQL**: Logical replication via "pgoutput" or "wal2json" plugins. Debezium captures the WAL and publishes to Kafka.

- **MySQL**: Binlog-based CDC via Debezium or AWS DMS.

CDC is the gold standard for correctness but adds operational complexity – you need a CDC consumer (Debezium, Kafka connector, or a managed service like Datastream on GCP).

Strategy 3 – Event-Driven (Change Feed / Streams)

NoSQL databases often provide native event streams:

- **Cosmos DB**: Change Feed provides an ordered and durable stream of document changes with near real-time updates and no polling required.

- **DynamoDB**: DynamoDB Streams captures the last 24 hours of changes and can be consumed using Lambda.

- **Firestore**: Snapshot listeners or Cloud Functions triggers can be used to capture changes.

- **Delta Lake**: Change Data Feed tracks row-level changes in Delta tables.

These are some of the most operationally efficient approaches for teams already using these data stores.

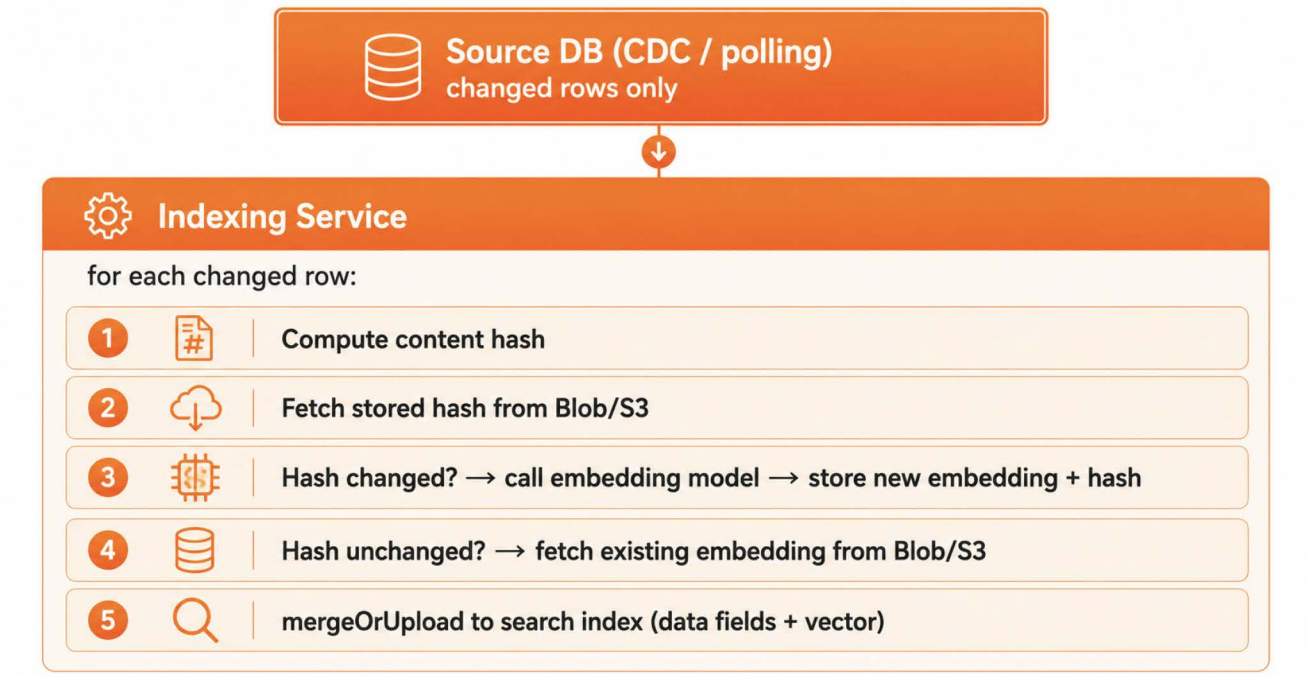
Strategy 4 – Content Hash Deduplication

Regardless of which change detection strategy feeds your pipeline, content hashing prevents unnecessary embedding regeneration:

```
def should_regenerate_embedding(doc: dict, stored_hash: str) -> bool:
    embeddable_fields = f"{doc['specialty']}|{doc['name']}|{doc['medical_group']}"
    current_hash = hashlib.sha256(embeddable_fields.encode()).hexdigest()
    return current_hash != stored_hash
```

Store the hash alongside the embedding (in Blob Storage, S3, or a dedicated column). On each indexing run, only records whose content hash changed incur an embedding API call. For datasets where most daily updates involve addresses or status changes rather than content changes, this approach can significantly reduce embedding costs.

Putting It Together – Incremental Indexing Pipeline

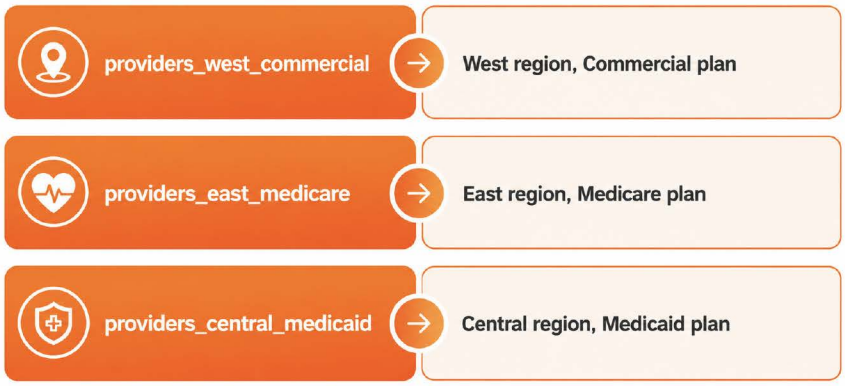


5. Ensuring Multi-Tenancy

Multi-tenancy in search means isolating data, queries, and sometimes infrastructure between different customers, plans, markets, or regions while running on shared or semi-shared infrastructure.

Strategy 1 – Index Per Tenant (Hard Isolation)

Each tenant gets a dedicated search index. Queries are routed to the tenant's index based on request context (JWT claims, API key, subdomain).



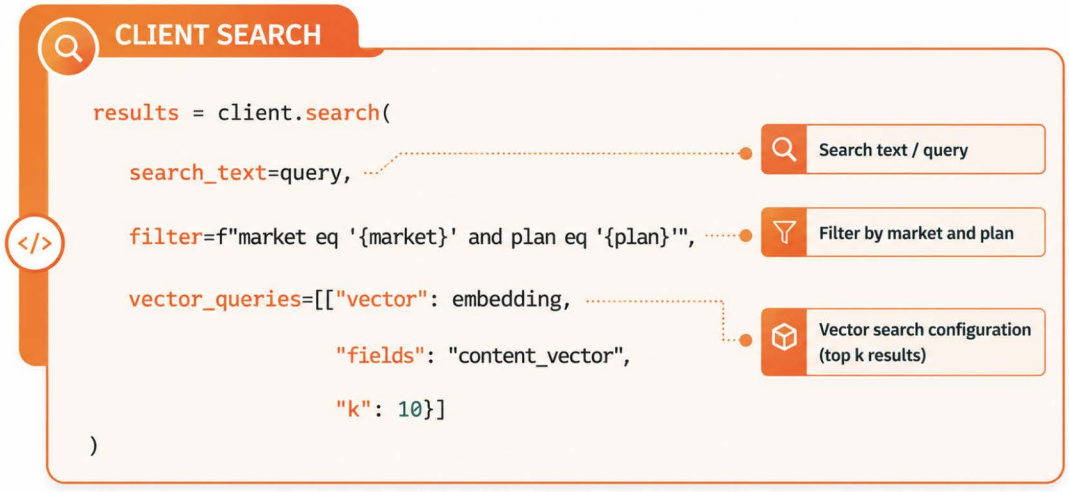
- Pros:
  - Complete data isolation.
  - Independent scaling.
  - Independent schema evolution.
  - Tenant deletion is a single index drop.

- Cons:
  - Index sprawl at high tenant counts (100+ tenants = 100+ indexes).
  - Higher management overhead.
  - Some platforms charge per index.

- Best for:
  - Regulated industries where data isolation is mandatory, or tenants with
  - significantly different data volumes and schema requirements.

Strategy 2 – Shared Index with Namespace Filters

All tenants share a single index. A tenant_id, market, region, or plan field is added to every document. All queries include a mandatory filter:



- Pros:
  - Simple, one index to manage.
  - No routing logic.
  - Works at any tenant count.
- Cons:
  - Vectors from all tenants participate in ANN search before filtering (pre-filter vs post-filter behavior differs by platform).
  - At very high tenant counts, a noisy tenant can affect query latency for others.
  - No isolation of sensitive data.
- Best for:
  - SaaS applications with many small tenants. Platforms like Qdrant that support native collection-level namespaces with efficient filtered vector search.

Strategy 3 – Hybrid (Recommended)

Segment by the highest-cardinality dimension into separate indexes, then filter within each index for lower-cardinality dimensions.

This balances index count management with meaningful isolation. The routing layer in your API selects the index based on the user's market, then applies plan/region filters within it.

Multi-Tenancy Beyond the Index

True multi-tenancy extends beyond data isolation to the full stack:

- **Auth:** JWT claims or API keys carry tenant context. RBAC enforced at the API gateway (APIM, Kong, AWS API Gateway).
- **Rate limiting:** Per-tenant rate limits at the gateway layer prevent one tenant from degrading another's experience.
- **Observability:** Metrics and logs tagged with 'tenant_id' allow per-tenant SLA monitoring.
- **Configuration:** Embedding models, filter schemas, and relevance tuning parameters stored per-tenant in a configuration store (Cosmos DB, DynamoDB, or a simple config table).

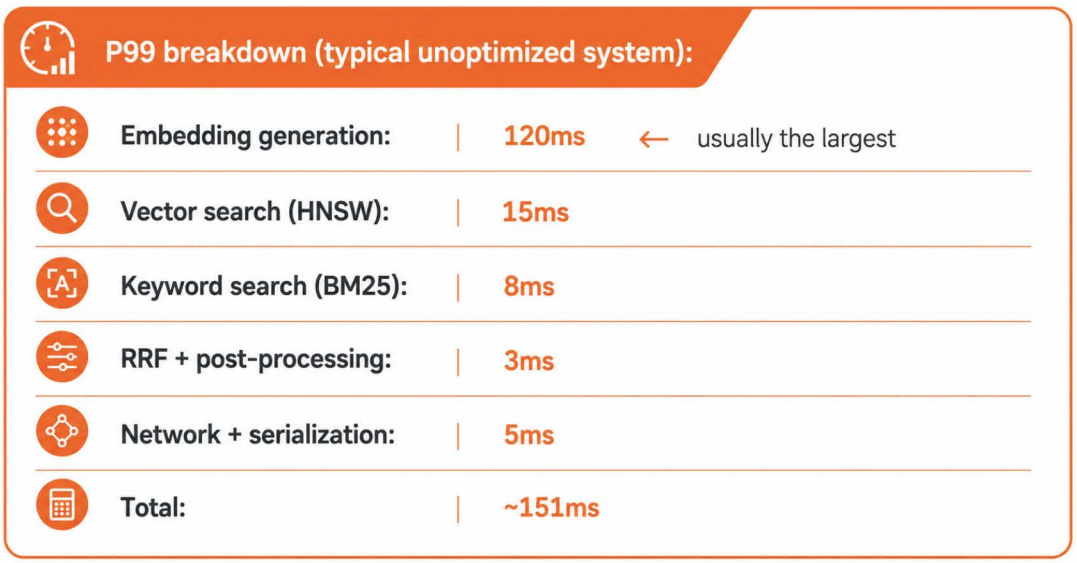
6. Approaches for Optimizing Latency

Identify the Bottlenecks First

Profile before optimizing. A typical hybrid search request touches:

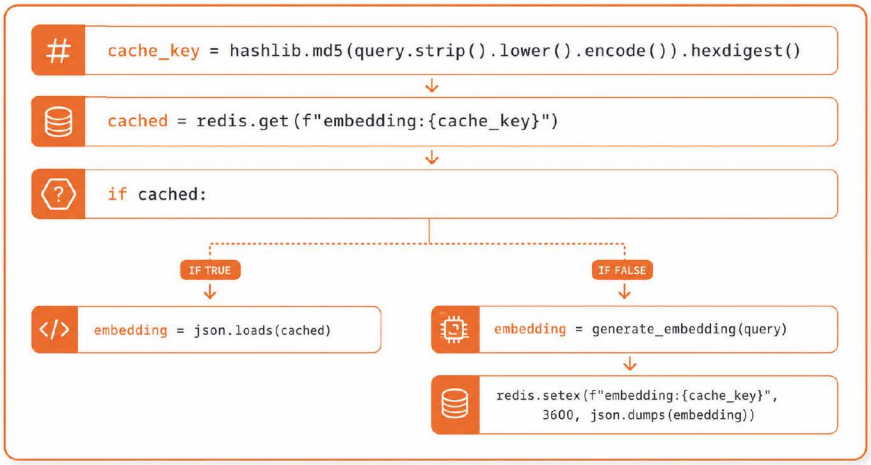
Embedding generation, vector ANN search, keyword search, RRF merge, optional re-ranking, and response serialization.

Each stage affects latency in different ways.



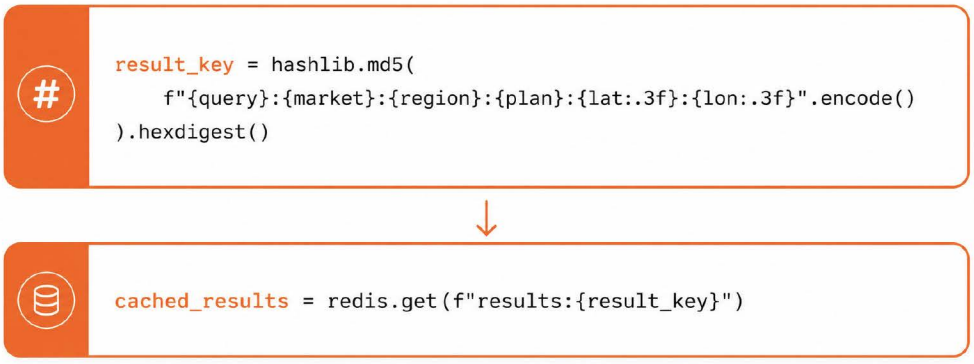
1 – Cache Embedding Generation

The user's query is often repeated or near-repeated. Cache embeddings for recent queries keyed by normalized query text:



2 – Cache Search Results

For high-traffic queries, cache the full result set. A search for “cardiologist near downtown” performed by 1,000 users on the same day produces the same result set, so it only needs to be computed once.



3 – Vector Quantization

Reduce memory footprint and increase throughput by quantizing float32 embeddings:

- **Scalar quantization (int8):** 4× memory reduction, ~95% recall with oversampling.
- **Binary quantization:** 32× memory reduction, ~90% recall. Works best with Matryoshka embedding models.

Both Azure AI Search and Qdrant support quantization natively. Quantized indexes fit in RAM on smaller machines, reducing ANN search latency.

4 – Reduce HNSW Search Space with Pre-Filtering

Rather than fetching k=100 vectors then applying geo and metadata filters, use indexed metadata fields to reduce the candidate set before vector search. Qdrant's payload indexing and Azure AI Search's pre-filter mode are designed for this. The tradeoff is reduced recall, as overly aggressive pre-filtering can exclude relevant results.

5 – Parallelize Retrieval

Run keyword search and vector search concurrently rather than sequentially:

</> PYTHON

```python
import asyncio

async def hybrid_search(query, embedding):
    keyword_task = asyncio.create_task(keyword_search(query))
    vector_task  = asyncio.create_task(vector_search(embedding))
    keyword_results, vector_results = await asyncio.gather(keyword_task, vector_task)
    return rrf_merge(keyword_results, vector_results)

```

6 – Tiered Re-ranking

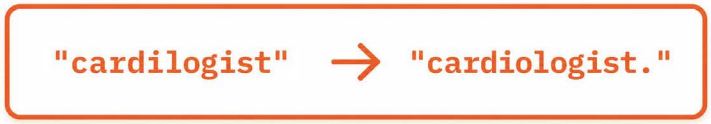
Reserve expensive cross-encoder re-ranking for the final top-20 candidates, not the full retrieval set. The retrieval stage handles recall and the re-ranker handles precision.

7. Approaches for Optimizing Search Quality

1 – Query Pre-Processing

Before the query hits retrieval, clean and normalize it:

- **Spell correction**: Symspell, Norvig's algorithm, or a fine-tuned seq2seq model.



- **Query expansion:** Expand acronyms and synonyms using a domain dictionary.

> "GP" → "general practitioner." "ED" → "emergency department" (not the other "ED").

- **Stop word handling:** Remove or downweight semantically empty terms.

> "I am looking for a doctor who treats" → "doctor treats."

- **Intent classification:** Route simple exact-match queries (NPI numbers, zip codes) to keyword-only path, bypassing the embedding model entirely.

2 – Embedding Model Selection

Not all embedding models are equal for your domain. General-purpose models (OpenAI text-embedding-3, E5-large) perform well across domains. Domain-specific models can outperform them significantly on in-domain queries:

- **Healthcare:** BioBERT, PubMedBERT, ClinicalBERT
- **Legal:** LegalBERT
- **Code:** CodeBERT, UniXcoder

Evaluate models using your own query-document pairs with metrics like NDCG@10 and MRR before committing.

3 – Relevance Tuning – Custom Scoring

After RRF produces a base ranking, apply domain-specific boosting:

</> PYTHON

```python
def apply_custom_scoring(results: list, context: dict) -> list:
    for r in results:
        r["final_score"] = r["rrf_score"]
        # Distance boost - closer providers rank higher when scores are similar
        r["final_score"] += (1 / (1 + r["distance_miles"])) * 0.1
        # Accepting patients boost
        if r["accepting_patients"]:
            r["final_score"] += 0.05
        # Rating boost for tie-breaking
        r["final_score"] += (r["rating"] / 5.0) * 0.03
    return sorted(results, key=lambda x: x["final_score"], reverse=True)
```

4 – Result Suppression

Filter out results that are semantically distant beyond a threshold, even if they appear in the top-K:

```
results = [r for r in results if r["vector_score"] > MINIMUM_SIMILARITY_THRESHOLD]
```

This prevents irrelevant results from appearing simply because the pool is sparse.

5 – LLM Re-rankers

Cross-encoder models (Cohere Rerank, BGE-Reranker-v2, Jina Reranker) take query-document pairs and produce a relevance score using full bidirectional attention – far more accurate than the independent encodings used for retrieval. Deploy as a post-retrieval step on the top 20–50 candidates. The latency cost is worth it for high-stakes queries.

6 – Offline Evaluation Pipeline

Build a golden test set of queries with known-relevant documents. Run retrieval against this set after every model or configuration change. Track NDCG@5, NDCG@10, MRR, and recall@K. Without an evaluation pipeline, relevance tuning is guesswork.

8. Pre-Requisites for Embarking on This Journey

Before committing to a hybrid search platform, teams often underestimate the foundational work required. The search index is only as good as what feeds it.

Data Readiness

Schema clarity: Know which fields are used for embeddings, such as specialty, description, and name, and which fields are used as filters, such as location, plan, language, and availability. Fields that mix both purposes need explicit handling decisions.

Data quality: Embedding inconsistent or poorly formatted data produces unreliable vectors. Run deduplication, normalization, and completeness checks before indexing. A provider specialty field that contains "Cardiologist," "cardiologist," "CARDIOLOGIST," and "Cardiology" will produce inconsistent embeddings.

Source of truth: Identify a single authoritative source for each entity type. Hybrid search built on conflicting data sources produces inconsistent results that are hard to debug.

Embedding Model Selection

Choose your embedding model before you begin. Changing models later requires re-embedding the entire corpus. Evaluate at least three candidate models on a sample of your real query-document pairs before committing.

Key evaluation criteria include recall for your query types, latency, cost per token, context window length, and multilingual support.

Infrastructure Decisions

Decide early on your managed-vs-selfhosted tradeoffs:

- **Managed (Azure AI Search, Vertex AI Search, Bedrock Knowledge Bases):** Lower ops burden, higher per-query cost, faster to get running, less control over HNSW parameters and quantization.
- **Self-hosted (Qdrant on Kubernetes, Elasticsearch on EC2):** Full control, lower at-scale cost, requires operational expertise in cluster management, monitoring, and upgrades.

Both are valid. The wrong answer is starting with one and migrating to the other at 100 million vectors.

Observability from Day One

Instrument the retrieval pipeline before it goes to production:

- **Query logging:** Log every query, the retrieved document IDs, latency, and cache hit/miss. This is gold for debugging and retraining.
- **Relevance feedback:** If users click on result #5 consistently while ignoring #1 and #2, that is a signal. Build a feedback loop.
- **Latency tracking:** P50, P95, P99 for each stage (embedding, ANN, keyword, RRF, re-rank). Alert on regressions.

- **Index health:** Track index size, embedding coverage (% documents with a vector), staleness (% documents not refreshed in N days).

Team Skills

Hybrid search at production scale requires a blend of skills that rarely coexist in a single person:

- Information retrieval fundamentals (BM25, TF-IDF, NDCG)
- Embedding model training and evaluation
- Distributed systems (for the indexing pipeline)
- API design and caching
- Data engineering (CDC, streaming, schema management)

Plan for this cross-functional requirement in team composition and knowledge transfer.

Cost Modelling

Run the numbers before you start:

- **Embedding generation cost:** Number of documents × average tokens per document × cost per 1K tokens × re-indexing frequency
- **Vector storage cost:** Number of documents × embedding dimensions × 4 bytes (float32) × storage cost per GB
- **Query cost:** Queries per day × embedding cost per query + search platform per-query cost (if applicable)
- **Re-ranking cost:** Queries routed to re-ranker × latency budget

> Content hashing (Section 4) and embedding caching (Section 6) are the two highest-leverage cost reduction techniques and should be built in from the start, not retrofitted.

Challenge Begins After Retrieval

Hybrid search combines keyword retrieval with semantic understanding to improve both relevance and precision. While implementation details differ by platform, the core pattern remains the same: embed, index, retrieve, merge, and rank results.

Production challenges usually begin after retrieval. Incremental indexing, latency, embedding costs, tenant isolation, and change detection are what determine whether a system can scale reliably. Retrieval is only one part of the problem. Operational architecture matters just as much.

The good news is that major cloud platforms already provide the core building blocks for hybrid search. The good news is that major cloud platforms already provide the core building blocks for hybrid search. Success depends on making the right architectural decisions early and building systems that can handle change reliably over time.

Build for the query your users will actually type, not the query you wish they would type.

Cross-Platform Component Reference

Each row is an architectural layer. Each column is a cloud platform or deployment flavor. Use this as a reference when selecting tools or designing Excalidraw architecture diagrams.

Layer	Azure	AWS	GCP	Databricks	Cloud-Agnostic
Source of Truth	Azure SQL / Cosmos DB	Amazon RDS / DynamoDB / S3	Cloud SQL / Firestore / BigQuery	Delta Lake tables	PostgreSQL / MySQL / MongoDB
Embedding Generation	Azure OpenAI (text-embedding-3)	Amazon Bedrock (Titan, Cohere)	Vertex AI (text-embedding-gecko, multilingual)	Databricks Model Serving (BGE, E5, custom)	HuggingFace Inference / OpenAI API
Embedding Persistence	Azure Blob Storage	Amazon S3	Cloud Storage	Delta Lake (vector column in table)	MinIO / S3-compatible object store
Full-Text Index (BM25)	Azure AI Search	Amazon OpenSearch Service	Vertex AI Search (formerly Enterprise Search)	Delta Lake FTS (preview) or external Elasticsearch	Elasticsearch / OpenSearch (self-hosted)
Vector Index (HNSW)	Azure AI Search (integrated, same service)	OpenSearch k-NN or Bedrock Knowledge Bases	Vertex AI Vector Search (ScaNN-based)	Databricks Vector Search (managed HNSW)	Qdrant / Pinecone / Weaviate
Indexing Pipeline	AI Search Indexer (pull) or Azure Container Apps (push)	AWS Lambda / AWS Glue / Amazon EventBridge	Cloud Dataflow / Cloud Functions	Delta Live Tables (DLT) streaming pipeline	Apache Airflow or Prefect
Change Detection / CDC	SQL Change Tracking / Cosmos DB Change Feed	DynamoDB Streams / RDS CDC / Debezium	Pub/Sub + Datastream (CDC from Cloud SQL)	Delta Lake Change Data Feed (CDF)	Debezium + Kafka (any relational DB)
Query Serving	Azure Container Apps + Azure API Management	AWS Lambda + Amazon API Gateway	Cloud Run + Apigee	Databricks Model Serving endpoint or external API	FastAPI on Kubernetes
Cache	Azure Cache for Redis	Amazon ElastiCache (Redis)	Memorystore (Redis)	External Redis	Redis (self-hosted)
Auth / Security	Azure Entra ID + Key Vault	AWS IAM + Secrets Manager	Google IAM + Secret Manager	Unity Catalog + Databricks Secrets	Keycloak / Auth0 + HashiCorp Vault

Notes: Azure AI Search provides BM25 + HNSW in a single service (rows 4 & 5 merge). Databricks Vector Search syncs directly from Delta via Change Data Feed – no custom indexing pipeline needed. GCP ScaNN excels at billion-scale throughput. For < 10M vectors, pgvector on Cloud SQL (GCP/AWS/Azure) can replace both full-text and vector rows. Cloud-agnostic stack runs identically on any cloud with no proprietary lock-in.

INDIUM

About Indium

We are an AI services company specializing in Agentic AI, Data & Analytics, Application Engineering, and Quality Engineering.

By combining autonomous AI systems, modern engineering, and data intelligence, we build innovative AI solutions that drive measurable business outcomes and long-term value.

With 5,000+ associates worldwide, Indium partners with Fortune 500 companies, Global 2000 enterprises, and leading technology firms in Financial Services, Healthcare, Manufacturing, Retail, and Technology. Our teams operate in North America, India, the UK, Singapore, Australia, and Japan, helping businesses stay ahead in an AI-first world.

USA	INDIA	UK	SINGAPORE
Cupertino | Princeton Toll-free: +1-888-207-5969	Chennai | Bengaluru | Mumbai Hyderabad | Pune Toll-free: 1800-123-1191	London Ph: +44 1420 300014	Singapore Ph: +65 6812 7888

www.indium.tech

For Sales Inquiries
sales@indium.tech

For General Inquiries
info@indium.tech